© Remo Markgraf, info@tnms.de, www.tnms.de

V1.5, 17.03.2025

Intro

Disabling and Enabling of interrupts is a common scenario in embedded SW. The Cortex-M offers suitable mechanisms for this, but the topic is not quite so trivial because of potentially occurring race conditions and therefore we take a deeper look at a process optimized for the Cortex-M.

Scenario

In a code segment, which can be a normal function, an interrupt service routine or a task in an RTOS context, all interrupt handlings should be temporarily suppressed to unlock them at a later time.

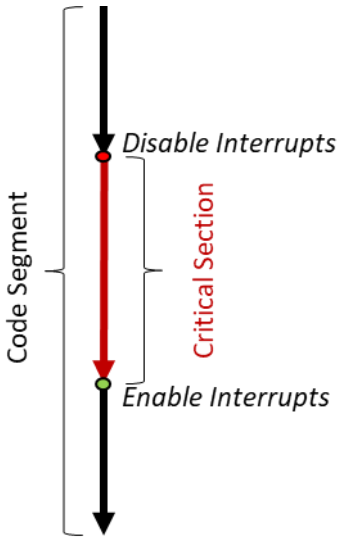

Figure 1: Critical Section

As shown in Figure 1, the code segment between the disable and the enable of interrupts forms a Critical Section that cannot be interrupted by exceptions or task switching of an RTOS.

If several such Critical Sections are nested by interrupts of different priorities or by semi-parallel running tasks of a multitasking operating system, then you have to ensure that at the end of a Critical Section the interrupts are only released again if they were not already disabled before the Critical Section.

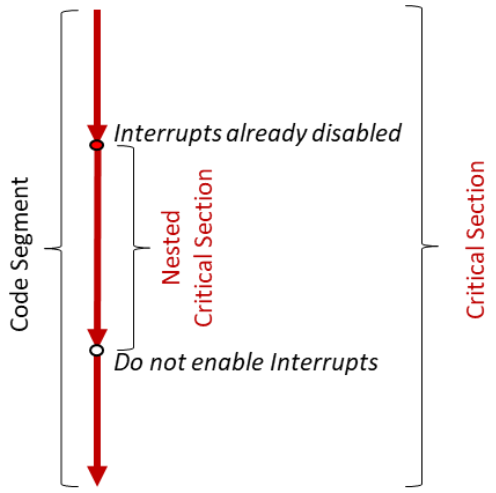

Figure 2: Nested Critical Section

Figure 2 shows such an example, which can happen, for example, whent a function is called within a Critical Section, which wants to execute a Critical Section on its own. Nested Critical Sections make sense, since this function could also be called from outside of a critcal section.

Race Conditions

When disabling interrupts, you have to remember, for example, in a local variable whether the interrupts were already disabled. At the end of the Critical Section, the interrupts are re-enabled, or not, according to the state stored in the local variable, before disabing.

Since the sequence of reading and remembering the state and disabling of interrupts is not possible within a single uninterruptible instruction, Race Conditions may occur here. This happens when the program flow is interrupted, for example, by a higher prior interrupt, exactly between reading and disabling, and the interrupts get disabled within this highest prior interrupt service routine. Then you would have read that the interrupts are not locked and release them again at the end of the Critical Section. In the higher priority interrupt, however, the interrupts have also been disabled, so releaseing them at the end of the Critical Section would be incorrect and thus change the system behavior and cause errors. As a matter of fact disabling of interrupts does not properly work like that!

To avoid such race conditions Critical Sections are usually implemented by means or semaphores or mutexes. The Cortex-M offers wonderful exclusive primitive LDREX, STREX, ... with the help of which semaphore and mutexes can be realized without having to disable interrupts.

In our scenario there is a simpler way to solve the problem. Let's see how it looks like….

Solution proposal

We can use a primitive counting semaphore that does not need access protection as long as we only access it within the Critical Section. I.e. we can simply use a variable to count the number of dis- and enabling. Anyway, an access violation is already avoided due to the Critical Section.

So we do not use the PRIMASK bit realized in the Cortex-Mx core HW to remember whether the interrupts were blocked, but a simple counter. **Disable = Increment**, **Enable = Decrement** and only if the counter reaches zero, the PRIMASK bit is reset in the HW and thus the interrupts are released again. In the code snippet in Figure 3, we see that we of course have to set the PRIMASK bit before we increment the counter in order to start the Critical Section, which automatically protects the counter.

Of course, all code segments that dis- or enable interrupts must use the same counter.

```
1     #include "stm32f10x.h"
2
3     static volatile uint32_t disable_counter = 0;
4
5     void disable_Interrupts( void )
6     {
7           __disable_irq();
8           // START OF CRITICAL SECTION
9           disable_counter++;
10    }
11
12    void resume_Interrupts( void )
13    {
14          //may only be called from within a critical section
15          disable_counter--;
16          if( disable_counter == 0UL )
17          {
18                __enable_irq();
19                // END OF CRITICAL SECTION
20          }
21    }
```

Figure 3: Code example

This can be achieved with a global variable. Clean Code rules teach us to apply the principle of encapsulation and thus to avoid global variables wherever possible. We therefore pack the functions for disabling and releasing the interrupts into an own C or C++ module and can then realize the counter as a static variable in this module. The `volatile` keyword ensures that the compiler generates code that does not cache the content of the counter in a core register and thereby always uses the actual value from the memory. In figure 3 we see a possible realization of such a module.

Of course, we should protect the code against incorrect calls. To prevent the incorrect call of the `resume_Interrupts()` function out of a non-critical section. then we can put the entire function body into a condition that is only executed if the counter is bigger than zero. This prevents the counter underflow from zero to the maximum uint32t value. Alternatively, you can precede the decrement (line 15 in figure 3) with a call to an assert macro:

```
assert(__get_PRIMASK() != 0);
```

The `__get_PRIMASK()` function returns zero if the interrupts are enabled. The assert macro executes an exit if the value does not correspond to the expectation. In addition to underflow, the counter can also overflow when increment. We use a 32-bit counter here, so an overflow is not one of the likely scenarios. While saving memory space during refactoring it might easily be changed into a 8-bit counter and then it is good if an assertion protects us from surprises. The following additional assert statement after increment performs this job.

Thanks to Mr Hofbauer for the proposed improvement.

```
assert(disable_counter != 0);
```

Finally, here are some additional hints:

- In order to compile the assert calls error-free, you have to insert the line #include <assert.h> at the top of the module.
- The assert macro can be switched off by the global define NDEBUG. In the production code you do not necessarily want to run an exit(). Then the macro is simply ignored at the compile time and does not cost memory or computing time. The global define is set in the compiler settings or command line for most compilers.
- The reset, the non-maskable interrupt and also the hardfault handler are enabled even when interrupts are switched off, since they have negative priorities and thus have the highest priorities in the system. In the non-Maskable interrupt and in the hardfault handler, the interrupts should therefore not be disabled. It is not required, since they do not get handled due to the lower priority anyway.
- The setting of the PRIMASK bit affects the handling of all interrupts (except reset, non-Maskable interrupt and hardfault). However, the individual pending bits of the interrupts are still set if the respective interrupt is pended. These pending bits remain set until they are either reset by the HW when servicing the interrupt, or by setting the appropriate clear-pending bit by SW. After releasing Interrupts, all interrupts that occurred during the Critical Section will be handled in order of their priority. There is no HW counter for the number of the occurrence of an interrupt during the Critical Section, it is just a single bit per interrupt.
- After a reset, the PRIMASK bit is cleared and the interrupts are enabled. The individual enable bits of the interrupts are not enabled after the reset, so only the hardfault, the NMI and the reset can occur.
- It is essential for the disable-counter function that all dis- and enabling of interrupts is done via the new functions that use the disable-counter. This ensures that the interrupts are enabled before the first call of the disable-interrupts() function, i.e. the PRIMASK bit is still reset. *Once more, thanks to Mr Hofbauer for the note.*
- The include statement in Figure 3 line 1 must of course be adapted for the respective controller you are using. The principle works for ALL Cortex-M, from M0 to M85.
- Please also refer to my blog on the topic of Exclusive Primitives LDREX, STREX, CLREX and their CMSIS C variants, which are briefly touched here.
- For any questions or proposals, please do not hesitate to contact me at info@tnms.de .

Have fun with Cortex-Mx 😊.

Finally, please find below two code examples for criticalsection.h and criticalsection.c

```
1   /*****************************************************************************//**
2   * @file criticalsection.h
3   * @version V1.1
4   * @date 14th of Feb 2025
5   * @author Remo Markgraf
6   * Copyright (C) 2025 Remo Markgraf info@tnms.de All rights reserved.
7   * @warning
8   * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS,
9   * IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, THE WARRANTIES OF
10  * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT TO
11  * THIS SOFTWARE. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY CLAIM,
12  * DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
13  * OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE
14  * USE OR OTHER DEALINGS IN THE SOFTWARE.
15  *****************************************************************************/
16  #ifndef CRICICALSECTION_H_
17  #define CRICICALSECTION_H_
18
19  #ifdef __cplusplus
20  extern "C" {
21  #endif
22
23  #include "stm32f10x.h"
24
25  //prototypes
26  void disable_Interrupts( void );
27  void resume_Interrupts( void );
28
29
30  #ifdef __cplusplus
31  }
32  #endif
33
34  #endif //CRICICALSECTION_H_
35
```

```c
/*****************************************************************************//**
* @file criticalsection.c
* @version V1.1
* @date 14th of Feb 2025
* @author Remo Markgraf
* Copyright (C) 2025 Remo Markgraf info@tnms.de All rights reserved.
* @warning
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS,
* IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, THE WARRANTIES OF
* MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT TO
* THIS SOFTWARE. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY CLAIM,
* DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
* OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE
* USE OR OTHER DEALINGS IN THE SOFTWARE.
*****************************************************************************/
#include <assert.h>
#include "criticalsection.h"

static volatile uint32_t diable_counter = 0;

void disable_Interrupts( void )
{
	__disable_irq();
	// START OF CRITICAL SECTION
	diable_counter++;
}

void resume_Interrupts( void )
{
	//may only be called from within a critical section
	assert(__get_PRIMASK() != 0);

	diable_counter--;
	if( diable_counter == 0UL )
	{
		__enable_irq();
		// END OF ALL CRITICAL SECTIONS
		assert(__get_PRIMASK() == 0);
	}
}

```

Hi, my name is Remo Markgraf and I train and consult extensively in projects related to Cortex-M, embedded software development and testing, test-driven development and agile development for embedded systems. Training and Consulting is provided in German and English language. My experience covers numerous areas of software development, test engineering, system architecture, project, product, lifecycle and business management and of course Live und Online Trainings www.tnms.de .